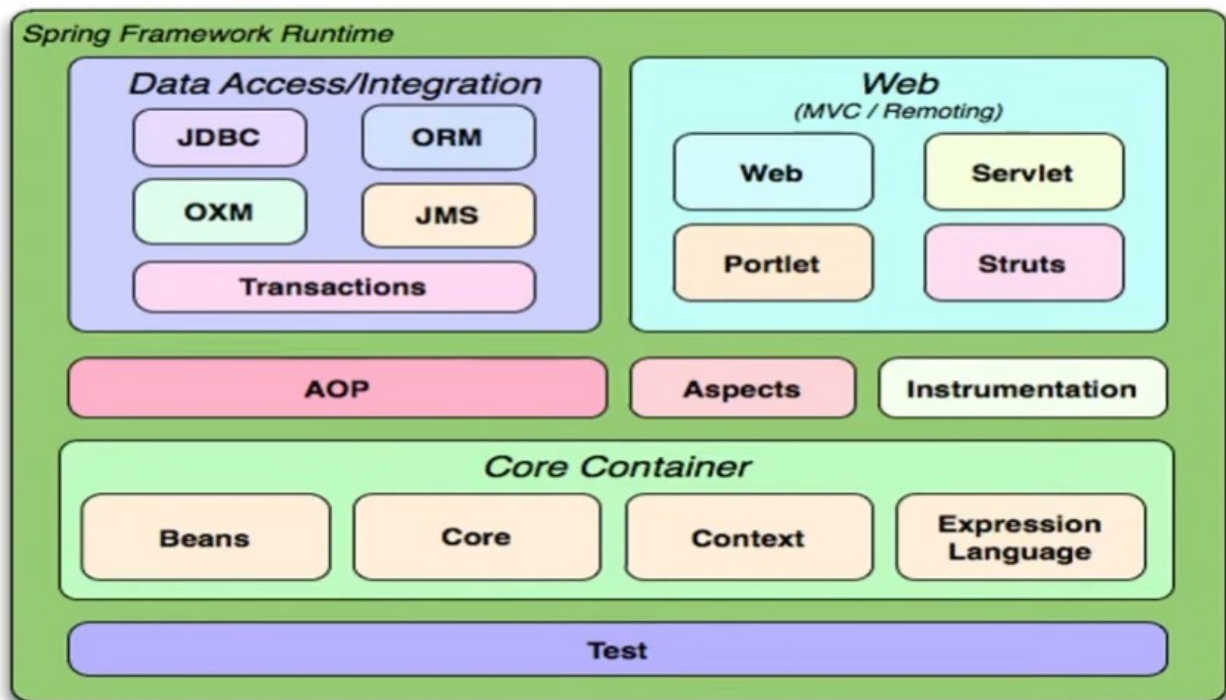


Spring Framework

Framework open source desenvolvido para a plataforma Java baseado nos padrões de projetos inversão de controle e injeção de dependência.

Sua estrutura é composta por módulos afins de reduzir a complexidade no desenvolvimento de aplicações simples ou corporativas.

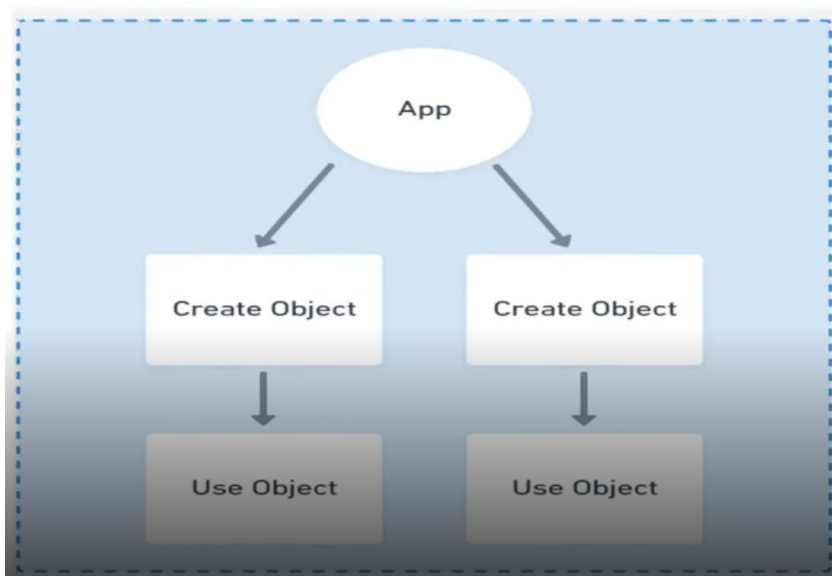
Módulos



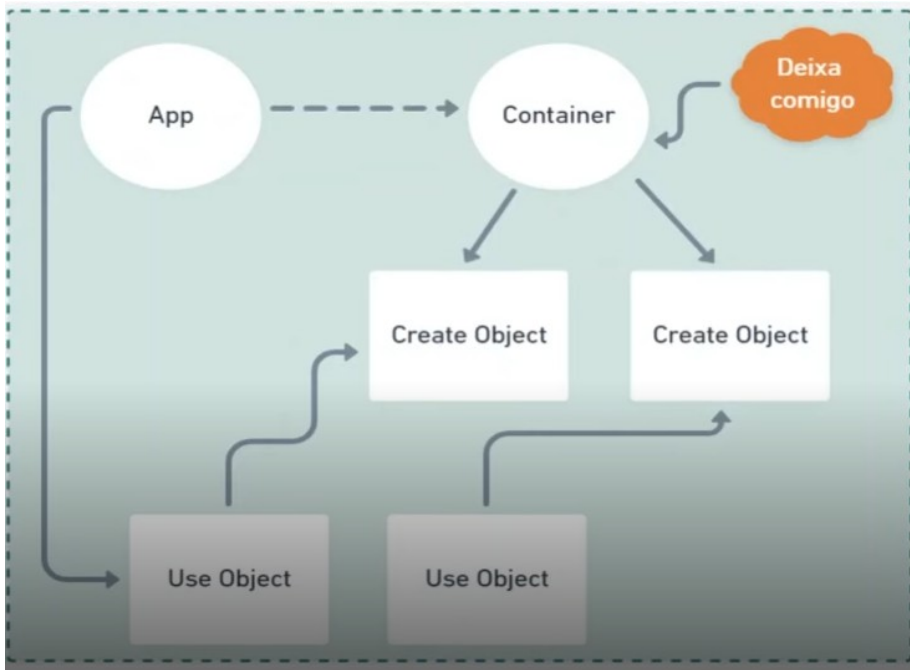
Inversão de controle ou IOC

Trate-se do redirecionamento do fluxo de execução de um código retirando parcialmente o controle sobre ele e delegando-o para um contêiner, minimizando o acoplamento do código.

SEM IOC:

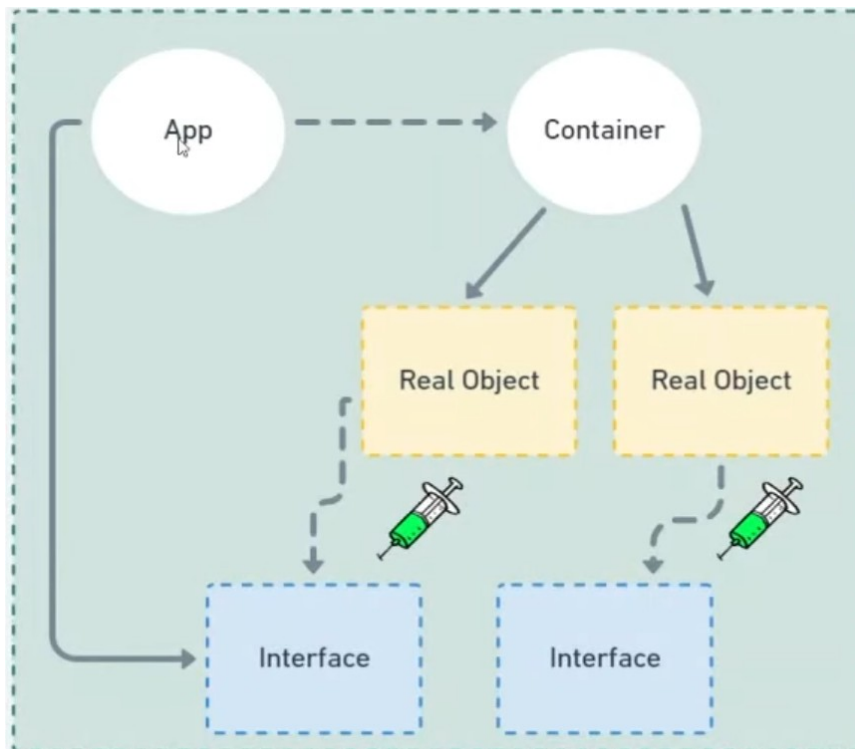


Com IOC:



Injeção de dependências

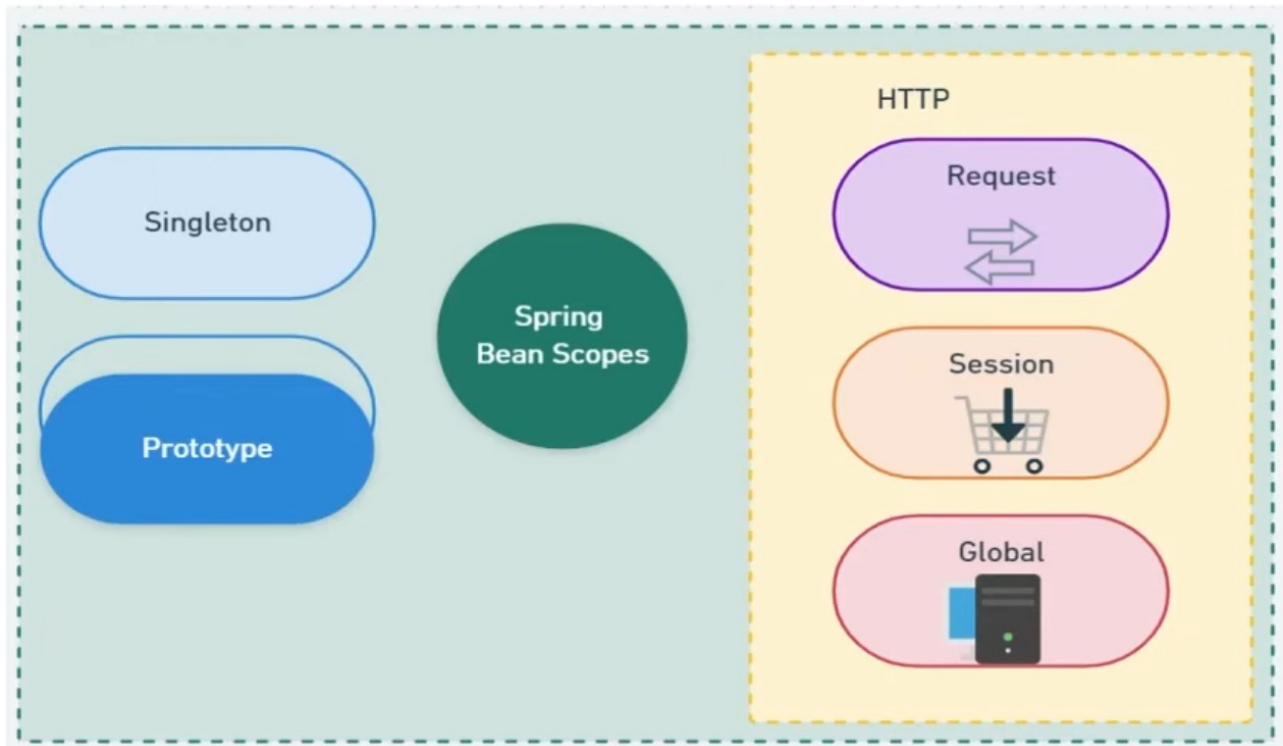
É um padrão de desenvolvimento com a finalidade de manter baixo o nível de acoplamento entre módulos de um sistema.



Beans

Objeto que é criado, montado e gerenciado por um contêiner através do princípio da IOC.

Scopes



Singleton: O contêiner do Spring IOC define apenas uma instância do objeto para toda a aplicação;

Prototype: Será criado um novo objeto a cada solicitação ao contêiner;

HTTP –

Request: Um bean será criado para cada requisição HTTP. Os objetos existirão enquanto a requisição estiver em execução;

Session: Um bean será criado para cada sessão de usuário;

Global: Cria um bean para o ciclo de vida do contexto da aplicação.

AutoWired

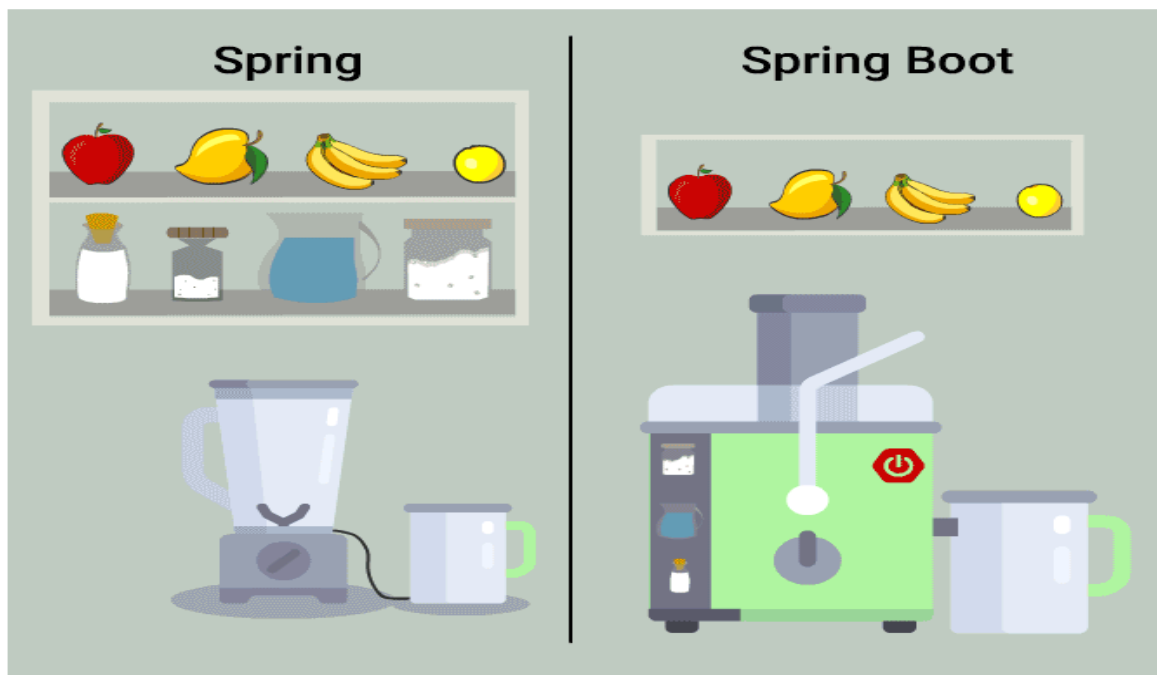
Uma anotação onde deverá ocorrer uma injeção automática de dependência.

Springboot

Enquanto que o Spring Framework é baseado no padrão de injeção de dependências, o Springboot foca na configuração automática.

Antes do SpringBoot

- Dependência individual;
- Verbosidade;
- Incompatibilidade de versões;
- Complexidade de gestão;
- Configurações complexas e repetitivas.



Dado que a maior parte das configurações necessárias para o início de um projeto são sempre as mesmas, por que não iniciar um projeto com todas estas configurações já definidas?

Starters

```
<dependencies>
  <!-- Spring -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${springframework.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${springframework.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${springframework.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${springframework.version}</version>
  </dependency>
  <!-- Hibernate -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
  </dependency>
  <!-- MySQL -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${mysql.connector.version}</version>
  </dependency>
  <!-- Joda-Time -->
  <dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>${joda-time.version}</version>
  </dependency>
  <!-- To map JodaTime with database type -->
  <dependency>
    <groupId>org.jadira.usertype</groupId>
    <artifactId>usertype.core</artifactId>
    <version>3.0.0.CR1</version>
  </dependency>
</dependencies>
```

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.4.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <!-- MySQL -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${mysql.connector.version}</version>
  </dependency>
</dependencies>
```

- Coesão;
- Versões compatíveis;
- Otimização de tempo;
- Configurações simples;
- Foco no negócio.

Alguns Starters

Spring-boot-starter-*

- **data-jpa**: Integração ao banco de dados via JPA - Hibernate.
- **data-mongodb**: Interação com banco de dados MongoDB.
- **web**: Inclusão do container Tomcat para aplicações REST.
- **web-services**: Webservices baseados na arquitetura SOAP.
- **batch**: Implementação de JOBS de processos.
- **test**: Disponibilização de recursos para testes unitários como JUnit
- **openfeign**: Client HTTP baseado em interfaces
- **actuator**: Gerenciamento de monitoramento da aplicação.

Quando usar component ou bean?

Component: Componentes que serão escaneados na aplicação. Utilizar quando tiver acesso ao código fonte.

Bean: Utilizar quando não tiver acesso ao código fonte.

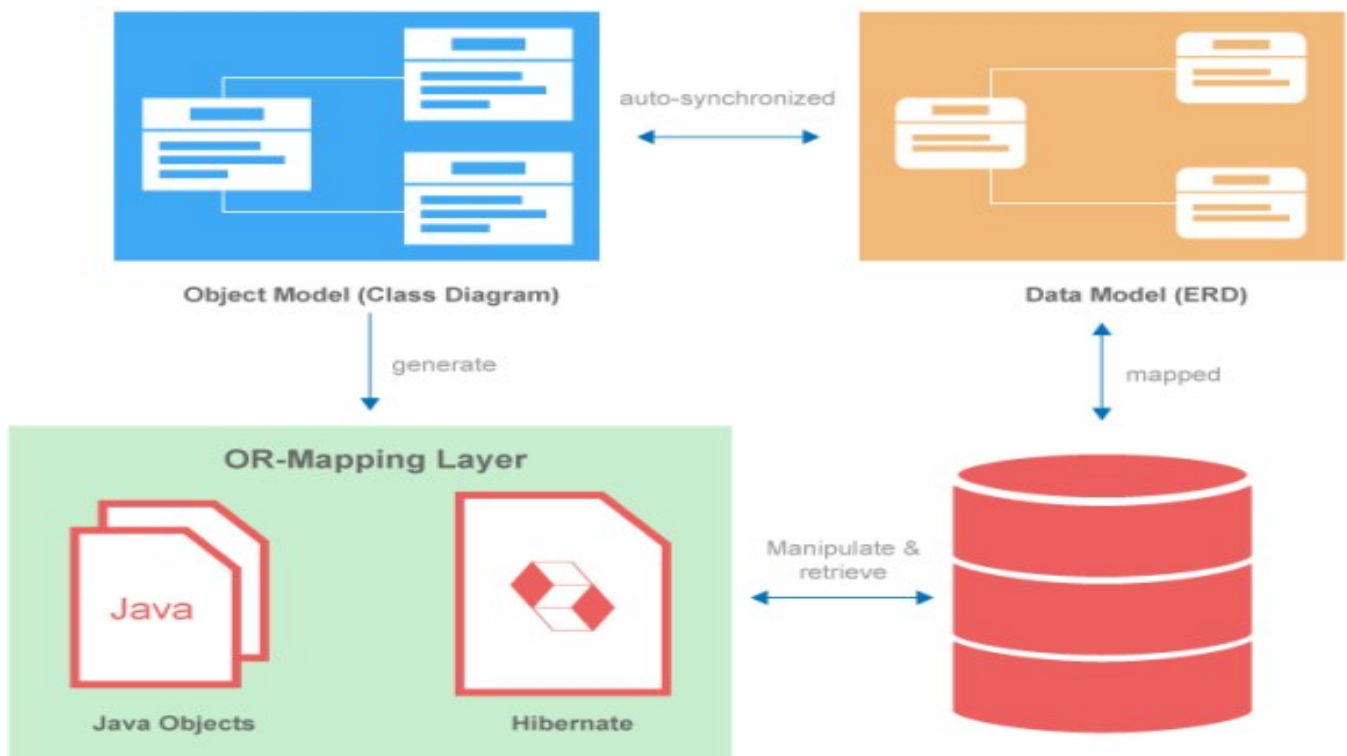
Nem tudo é =

- `Application.properties`: Mantém estados, informações e propriedades de um contexto da aplicação que garantimos que não haverá alteração no decorrer da execução.
- `@Value`:

Conceitos de ORM e JPA

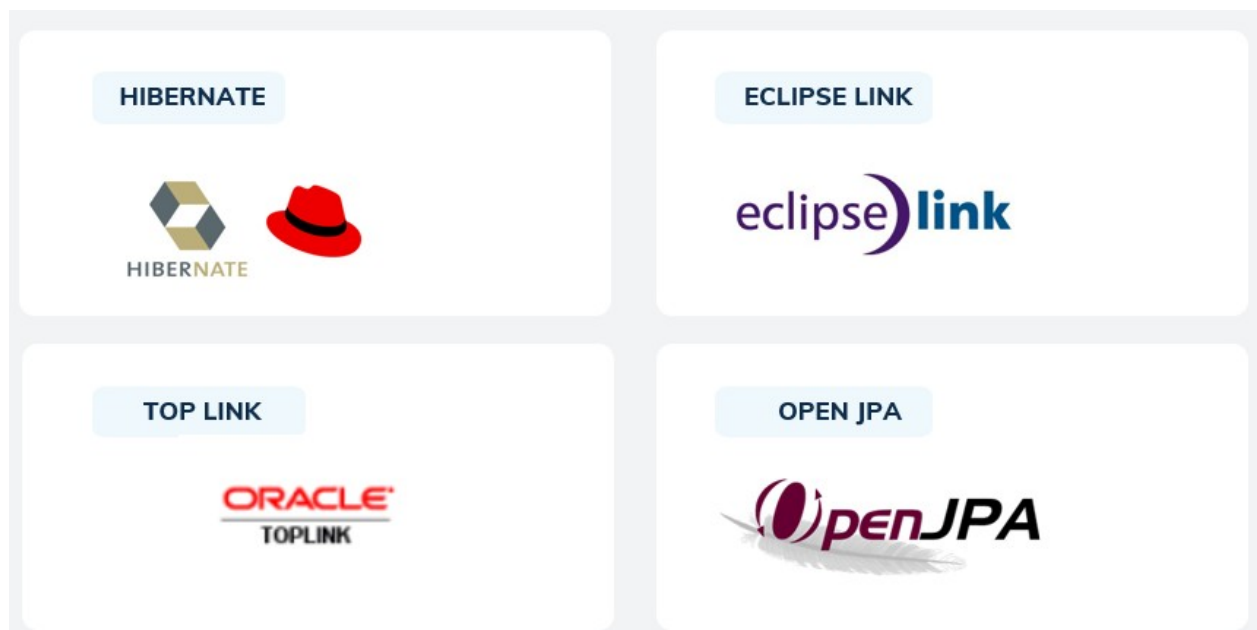
O que é ORM?

Object-Relational Mapping é um recurso para aproximar o paradigma da POO ao contexto de BD relacional. O uso de ORM é realizado através do mapeamento de objeto para uma tabela por uma biblioteca ou framework.



JPA

Consiste em uma especificação baseada em interfaces, que através de um framework realiza operações de persistência de objetos em Java.



Mapeamento

Vamos conhecer os aspectos das anotações de mapeamento do JPA

- Identificação;
- Definição;
- Relacionamento;
- Herança;
- Persistência.

Mapeamento na prática:

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="tb_usuario")
public class Usuario {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="id_usuario")
    private Long id;

    private String nome;

    @Column(name="login_usuario")
    private String login;

    @Column(name="senha_usuario")
    private String senha;

}
```

Spring Data JPA

O Spring Data JPA adiciona uma camada sobre o JPA. Isso significa que ele usa todos os recursos definidos pela especificação JPA, especialmente o mapeamento de entidades e os recursos de persistência baseado em interfaces e anotações. Por isso, o Spring Data JPA adiciona seus próprios recursos, como uma implementação sem código do padrão de repositório e a criação de consultas de banco de dados a partir de nomes de métodos.

Interfaces

- CrudRepository
- JpaRepository
- PagingAndSortingRepository

Anotações

- @Query
- @Param

O Poderoso Repository

Repository Pattern

Repository é um padrão de projeto similar ao DAO (Data Access Object) no sentido de que seu objetivo é abstrair o acesso a dados de forma genérica a partir do seu modelo.

Principais métodos que já são disponibilizados pelo framework:

- **save**: Insere e atualiza os dados de uma entidade.
- **findById**: Retorna o objeto localizado pelo seu ID.
- **existsById**: Verifique a existência de um objeto pelo ID informado, retornando o boolean.
- **findAll**: Retorna uma coleção contendo todos os registros da tabela no banco de dados.
- **delete**: Deleta um registro da respectiva tabela mapeada do banco de dados.
- **count**: retorna a quantidade de registros de uma tabela mapeada no banco de dados.

Consultas Customizadas

Existem duas maneiras de realizar consultas customizadas, uma é conhecida como **QueryMethod** e a outra é **QueryOverride**.

Query Method

O Spring Data JPA se encarrega de interpretar a assinatura de um método (nome + parâmetros) para montar a **JPQL** correspondente.

Veja o exemplo de uma entidade que possui um endereço de e-mail e sobrenome e gostaria de filtrar por estes dois atributos.

```
public interface UserRepository extends Repository<User, Long> {  
  
    List<User> findByEmailAddressAndLastname(String emailAddress, String  
    lastname);  
  
}
```

Table 3. Supported keywords inside method names

Keyword	Sample	JPQL snippet
Distinct	findDistinctByLastnameAndFirstname	select distinct ... where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is , Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	... where x.age >= ?1

Query Override

Vamos imaginar que você precisará montar uma query um tanto avançada mas ficaria inviável utilizar o padrão **QueryMethod** ? Como nossos repositórios são interfaces não temos implementação de código, é aí que precisa definir a consulta de forma manual através da anotação **@Query**.

Os dois métodos realizam a mesma instrução SQL, consultando os usuários pelo seu campo **name** comparando com o perador **LIKE** do SQL.

```
public interface UserRepository extends JpaRepository<User, Integer> {  
    //Query Method - Retorna a lista de usuários contendo a parte do name  
    List<User> findByNameContaining(String name);  
  
    //Query Override - Retorna a lista de usuários contendo a parte do name
```

```

@Query("SELECT u FROM User u WHERE u.name LIKE %:name%")
List<User> filtrarPorNome(@Param("name") String name);

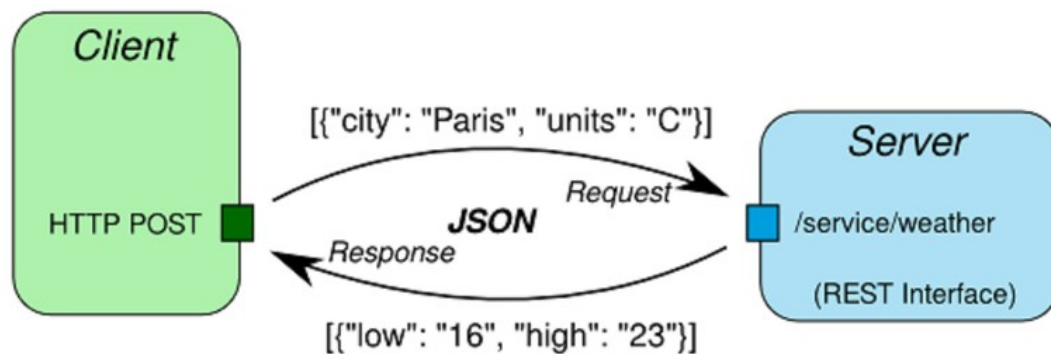
//Query Method - Retorna um usuário pelo campo username
User findByUsername(String username);
}

```

Introdução ao Spring Web

API:
Código

RESTful Web Service in Java



programável que faz a “ponte” de comunicação entre duas aplicações distintas.

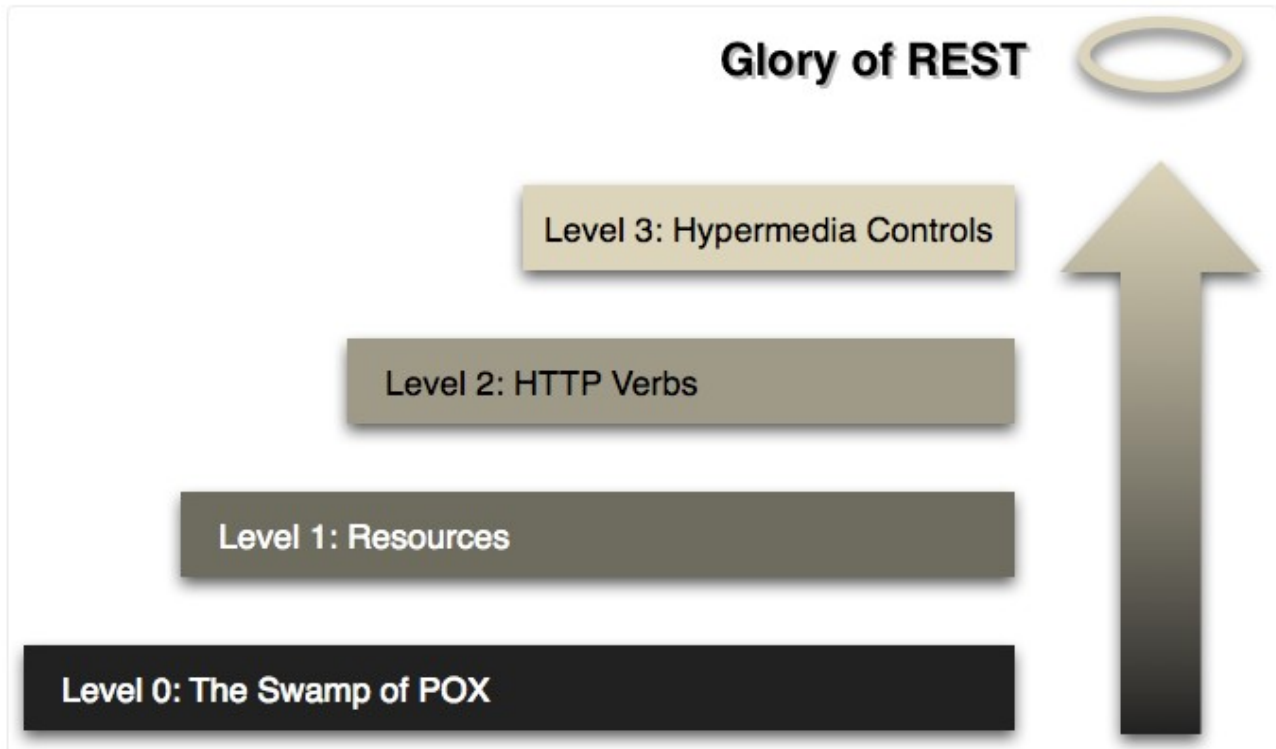
Rest e Restful: A API rest é como um guia de boas práticas e Restful é a capacidade de determinado sistema aplicar os princípios de Rest.

Princípios

Para que uma arquitetura seja RESTful, é necessário ter uma série de princípios ou padrões:

- **cliente-servidor:** significa aprimorar a portabilidade entre várias plataformas de interface do usuário e do servidor, permitindo uma evolução independente do sistema;
- **interface uniforme:** representa uma interação uniforme entre cliente e servidor. Para isso, é preciso ter uma interface que identifique e represente recursos, mensagens autodescritivas, bem como hypermedia (HATEOAS);
- **stateless:** indica que cada interação via API tem acesso a dados completos e compreensíveis;
- **cache:** necessário para reduzir o tempo médio de resposta, melhorar a eficiência, desempenho e escalabilidade da comunicação;
- **camadas:** permite que a arquitetura seja menos complexa e altamente flexível.

Nível de maturidade



Nível 0: Ausência de Regras

Esse é considerado o nível mais básico de uma API, quem implementa apenas esse nível não pode ser considerada REST pois não segue qualquer padrão.

Nível 1:	Verbo HTTP	URI	Operação
	POST	/getUsuario	Pesquisar Usuario
	POST	/salvarUsuario	Salvar
	POST	/alterarUsuario	Alterar
	POST	/excluirUsuario	Deletar

Aplicação de Resources

Observe que o nome dos recursos foram	Verbo HTTP	URI	Operação
	GET	/usuarios/1	Pesquisar Usuario
	POST	/usuarios	Salvar
	PUT	/usuarios/1	Alterar
	DELETE	/usuarios/1	Deletar

equalizados e para não gerar ambiguidade é necessário definir o verbo apropriadamente.

Nível 2: Implementação de verbos HTTP

Como a definição dos verbos já foi requisitada no Nível 1, o Nível 2 se encarrega de validar a aplicabilidade dos verbos para finalidades específicas como:

Verbo HTTP	Função
GET	Retorna Dados
POST	Grava dados
PUT	Altera Dados
DELETE	Remove Dados

Nível 3: HATEOAS

HATEOAS significa Hypermedia as the Engine of Application State. Uma API que implementa esse nível fornece aos seus clientes links que indicarão como poderá ser feita a navegação entre seus recursos. Ou seja, quem for consumir a API precisará saber apenas a rota principal e a resposta dessa requisição terá todas as demais rotas possíveis.

```
{
  "id": 1,
  "nome": "John",
  "sobrenome": "Doe",
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/clientes/1"
    },
    {
      "rel": "alterar",
      "href": "http://localhost:8080/clientes/1"
    },
    {
      "rel": "excluir",
      "href": "http://localhost:8080/clientes/1"
    }
  ]
}
```

No exemplo acima, podemos ver o resultado de uma API que implementa HATEOAS, veja que na resposta dessa API há uma coleção “links”, cada link aponta para uma rota dessa API. No caso desse exemplo, temos um link para a própria rota, um link para alterar um cliente e outra para excluir.

Springboot Rest API

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```

Controller

Um **Rest Controller** em Spring, nada mais é que uma classe contendo anotações específicas para a disponibilização de recursos HTTPs, baseados em nossos serviços e regras de negócio.

Anotações e configurações mais comuns:

- **@RestController**: Responsável por designar o bean de compoment, que surporta requisições HTTP com base na arquitetura REST.
- **@RequestMapping("prefix")**: Determina qual a URI comum para todos os recursos disponibilizados pelo **Controller**.
- **@GetMapping**: Determina que o método aceitará requisições **HTTP** do tipo **GET**.
- **@PostMapping**: Determina que o método aceitará requisições **HTTP** do tipo **POST**.
- **@PutMapping**: Determina que o método aceitará requisições **HTTP** do tipo **PUT**.
- **@DeleteMapping**: Determina que o método aceitará requisições **HTTP** do tipo **DELETE**.
- **@RequestBody**: Converte um JSON para o tipo do objeto esperado como parâmetro no método.
- **@PathVariable**: Consegue determinar que parte da URI será composta por parâmetros recebidos nas requisições.

Spring Security

Spring security é um grupo de servlet que ajudam a adicionar autenticação e autorização ao seu aplicativo web.

Terminologia

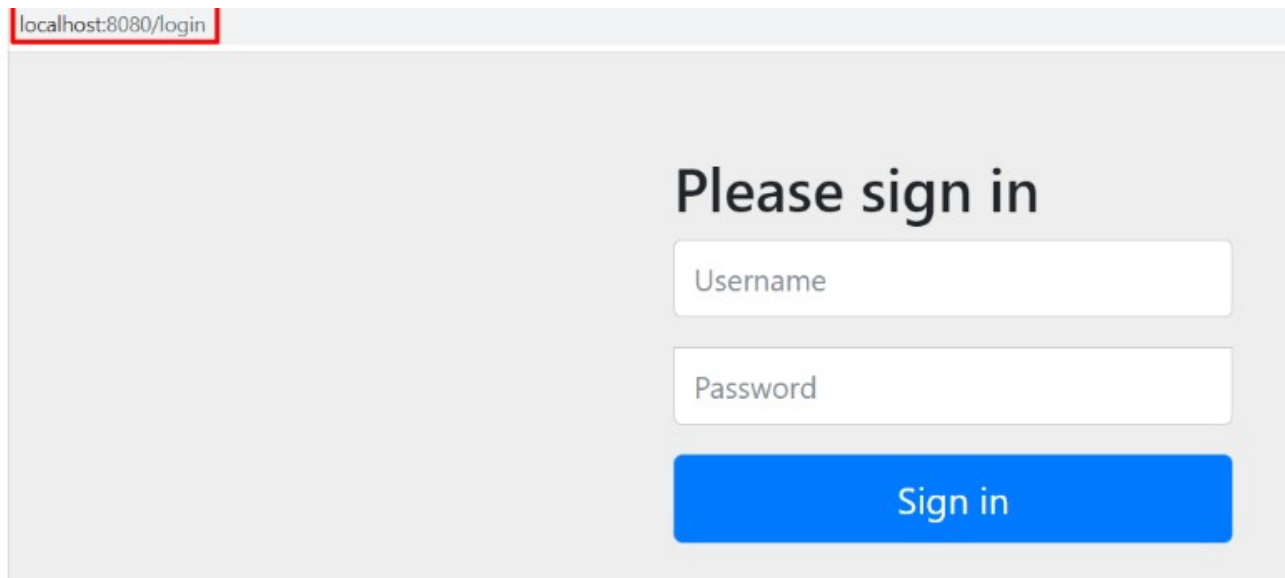
Autenticação: refere-se ao processo de verificação da identidade de um usuário, com base nas credenciais fornecidas. Um exemplo comum é inserir um nome de usuário e uma senha ao fazer login em um site. Você pode pensar nisso como uma resposta à pergunta: "*Quem é você?* " ;

Autorização: refere-se ao processo de determinar se um usuário tem permissão adequada para executar uma ação específica ou ler dados específicos, supondo que o usuário seja autenticado com êxito. Você pode pensar nisso como uma resposta à pergunta: "Um usuário pode fazer / ler isso? " ;

Princípio: refere-se ao usuário autenticado no momento;

Autoridade concedida: refere-se à permissão do usuário autenticado.

Função: refere-se a um grupo de permissões do usuário autenticado.



Autenticação Simples

O Spring possui algumas configurações para definir os usuários na sua camada de segurança.

Como sabemos por padrão o Spring Security habilita um usuário de nome user e gera uma senha aleatoriamente a cada inicialização. Para aplicações em produção esta não é uma abordagem um tanto aconselhável, e é por isso que vamos conhecer algumas outras configurações de segurança.

No application.properties

O Spring Security verifica se existe alguma configuração de segurança no arquivo **application.properties**.

```
spring.security.user.name=user  
spring.security.user.password=user123  
spring.security.user.roles=USERS
```

Em memória

Esta configuração permite criar mais de usuário e perfis de acesso.

É necessário criar uma classe que estenda `WebSecurityConfigurerAdapter` conforme abaixo:

```
import org.springframework.context.annotation.Configuration;  
import  
org.springframework.security.config.annotation.authentication.builders.Authenti  
cationManagerBuilder;  
import  
org.springframework.security.config.annotation.method.configuration.EnableGloba  
lMethodSecurity;
```

```

import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user")
            .password("{noop}user123")
            .roles("USERS")
            .and()
            .withUser("admin")
            .password("{noop}master123")
            .roles("MANAGERS");
    }
}

```

Configure Adapter

Nos exemplos acima já podemos considerar um nível de segurança em nossa aplicação, mas se percebermos o esforço de configuração para as novas funcionalidades, poderá não ser uma abordagem tão satisfatória. Para isso, vamos tentar deixar a parte de configuração, centralizada na nossa **WebSecurityConfig**, removendo configurações adicionais em nossos **controllers**.

```

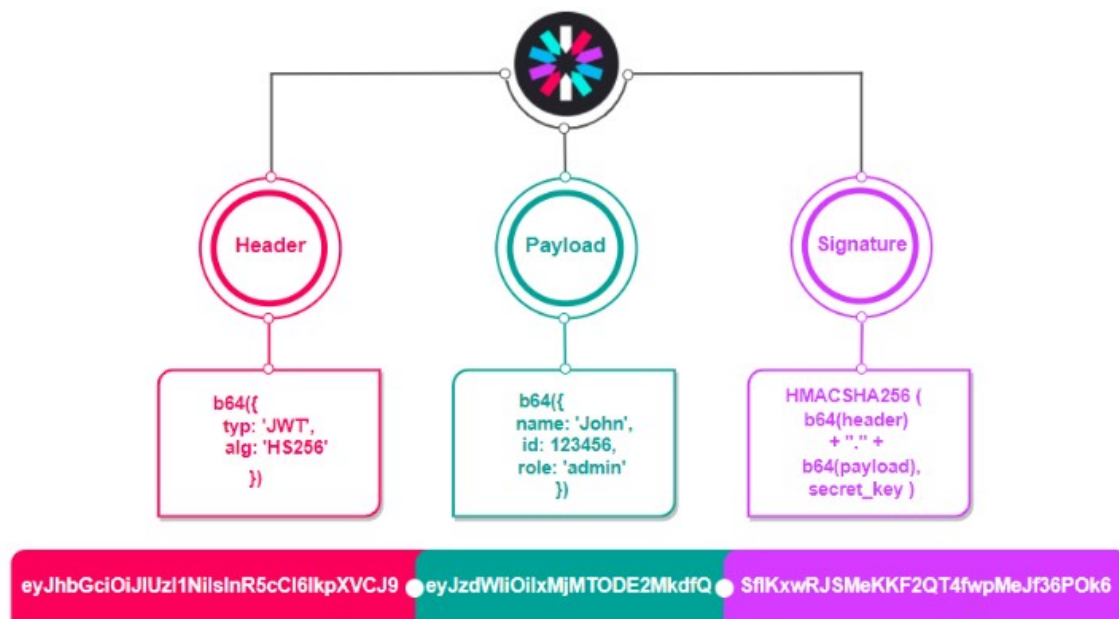
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/login").permitAll()
            .requestMatchers("/admins").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/users/**").hasAnyRole("USER", "ADMIN")
            .requestMatchers(HttpMethod.POST, "/users/**").hasRole("ADMIN")
            .requestMatchers(HttpMethod.PUT, "/users/**").hasRole("ADMIN")
            .requestMatchers(HttpMethod.DELETE, "/users/**").hasRole("ADMIN")
            .requestMatchers("/").hasAnyRole("ADMIN", "USER")
            .anyRequest().authenticated()
        )
        .csrf(AbstractHttpConfigurer.disable())
        .formLogin(Customizer.withDefaults()).logout(logout -> logout.logoutUrl("/logout"))
        .httpBasic(Customizer.withDefaults());

    return http.build();
}

```

JWT - JSON Web Token

O JSON Web Token - JWT é um padrão da Internet para a criação de dados com assinatura opcional e/ou criptografia, cujo conteúdo contém o JSON que afirma algum número de declarações. Os tokens são assinados usando um segredo privado ou uma chave pública/privada.



Qual a estrutura do JWT?

JWT é uma representação dividida em 03 partes:

- Header
- Payload
- Signature

Portanto, um JWT normalmente se parece com o seguinte: `xxxxx.yyyyy.zzzzz`.

Header

O header ou cabeçalho normalmente consiste em duas partes: o tipo de token, que é JWT e o algoritmo de assinatura que está sendo utilizado, como HMAC SHA256 ou RSA.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload

De fato, a estrutura do corpo contendo as informações de autenticação e autorização de um usuário.


```
{
  "sub": "pedro",
  "name": "PEDRO ARTHUR",
  "roles": ["USERS","ADMINS"]
}
```

Signature

Para criar a parte da assinatura, você deve pegar o cabeçalho codificado, o payload codificado, a chave secreta, o algoritmo especificado no cabeçalho e assiná-lo.

